Amged Alamari
CIS 4130
Professor Holowczak
11 September, 2024

Proposal: Analysis of New York City Taxi Trip Data (2010 - 2020)

Dataset Description:
For this project, I propose using the New York City Taxi Trip dataset, a free and open-source collection provided by the NYC Taxi and Limousine Commission (TLC). The dataset contains records of millions of taxi trips in New York City from 2010 to 2020, covering various trip attributes, including pick-up and drop-off times/locations, trip distance, fare amount, and payment method. The dataset has been extensively used in research projects focused on urban mobility and transportation analysis.

URL/Location for Downloading the Data:
The dataset is hosted on the NYC TLC website and can also be accessed through Google BigQuery. The dataset is available as CSV files, with a total size exceeding 50 GB.

Dataset Attributes (Columns):
The key columns include:

- **VendorID**: Taxi company code.
- **Pickup_datetime**: Start time of the trip.
- **Dropoff_datetime**: End time of the trip.
- **Passenger_count**: Number of passengers.
- **Trip_distance**: Distance traveled (miles).
- **RateCodeID**: Rate type for the trip.
- **Pickup_longitude/Pickup_latitude**: Pickup GPS coordinates.
- **Dropoff_longitude/Dropoff_latitude**: Drop-off GPS coordinates.
- **Payment_type**: Payment method used.
- **Fare_amount**: Fare charged for the trip.
- **Tip_amount**: Tip provided.
- **Total_amount**: Total trip cost (including surcharges).

Objective and Prediction Plan:
The goal of this project is to predict the amount of the tip based on trip attributes such as distance, passenger count, and pickup/drop-off locations. Using regression models, I will forecast the total tip amount. I plan to start with a linear regression model to explore potential relationships between variables. If the linear model does not perform well, I will experiment with advanced techniques like random forests or gradient boosting to capture more complex interactions and non-linearities in the data.

Impact:
Accurate tip predictions can improve fare estimation tools for customers and optimize route

planning for drivers. Insights from the model could also inform transportation policies and urban mobility analysis, identifying patterns in taxi usage and the effects of external factors such as weather and time of day.

Project Milestone 2

Created Google Cloud Storage Bucket:

- Bucket Name: **my-bigdata-project-aa**
- Folder: **/landing** for storing downloaded files.

Data Scraping Process:
- Used requests library to scrape trip data from a government website: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.
- Downloaded .parquet files for the years 2020-2023, iterating through months.

```
import requests

Headers = {'user-agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'}

base_url = 'https://d37ci6vzurychx.cloudfront.net/trip-data/'


years_list = ['2020', '2021', '2022', '2023']

month_list = ['01','02','03', '04', '05', '06', '07', '08','09', '10', '11', '12']


for year in years_list:

    for m in month_list:

        file_name = f"fhvhv_tripdata_{year}-{m}.parquet"

        full_url = base_url + file_name

        with open(file_name, 'wb') as outfile:

            response = requests.get(full_url, headers=Headers)
```
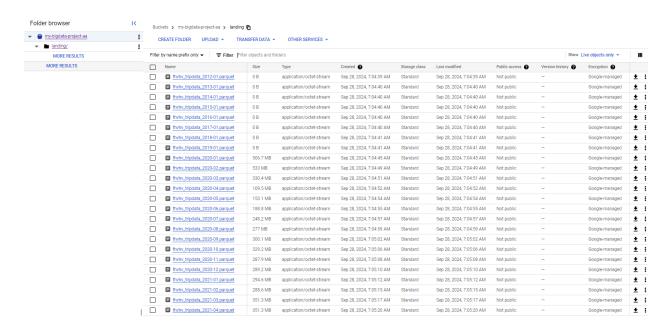
```
    if response.ok:

        outfile.write(response.content)
```

File Transfer to Bucket:

Used gcloud auth list and ls -l to gain permissions.
Moved .parquet files to the bucket's /landing folder using this code:

gsutil mv *.parquet gs://my-bigdata-project-aa/landing/

Project Milestone 3

EDA Objectives

The goal of the Exploratory Data Analysis (EDA) was to gain a better understanding of the dataset by visualizing key patterns and identifying relationships between variables. This helped in detecting potential issues and outliers that could inform the data cleaning process.

Key steps:

1. Descriptive Statistics: Calculated summary statistics such as mean, median, and standard deviation to understand the data distribution.
2. Data Visualizations:

- Histogram for Trip Distance: Showed how trip distances are distributed across the dataset.
- Scatter Plot for Fare Amount vs. Trip Distance: Illustrated the relationship between trip length and fare cost, highlighting trends and anomalies.
- Correlation Heatmap: Visualized correlations between numeric variables to identify strong relationships or unexpected patterns.

EDA Findings:

- Trip Distance: The majority of trips were short, with a sharp decline for longer distances, suggesting typical urban travel patterns.
- Fare Patterns: There was a clear, positive relationship between trip distance and fare amount, with longer trips generally costing more. However, some anomalies suggested inconsistent pricing.
- Correlations: Strong correlations were found between fare-related variables, such as fare_amount, extra, and tip_amount. The heatmap helped in identifying factors that affect fare prices.

Data Cleaning

Cleaning Objectives:

- Removing or imputing missing values.
- Ensuring consistent data types.
- Renaming columns for clarity and standardization.
- Dropping unnecessary columns.
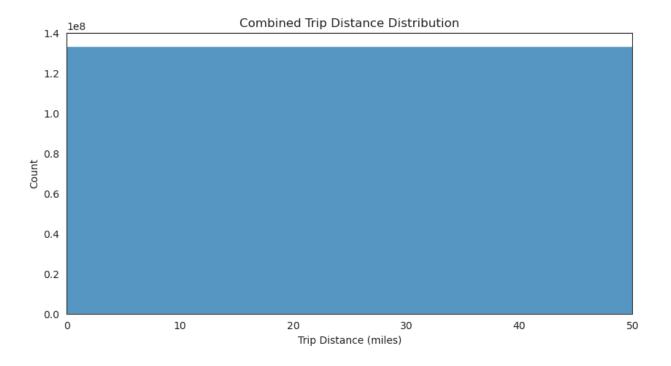
Implementation:

Python libraries such as Pandas were used to automate the cleaning process. This allowed for efficient handling of large datasets and ensured that the cleaning procedures were reproducible and scalable.
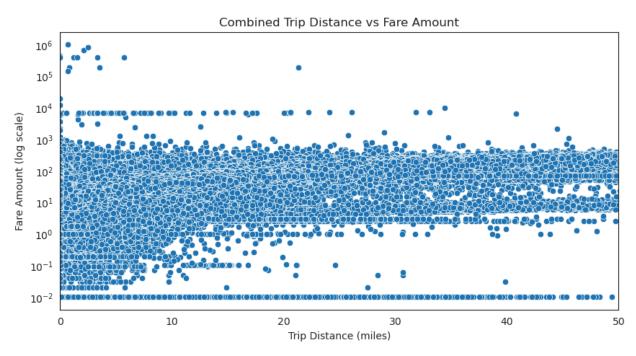
Results and Conclusion

Key Achievements:

- Consistent, Clean Dataset: The data was cleaned, standardized, and made consistent across all fields, enabling more reliable analysis.
- Automation: Automated the cleaning and processing tasks, making it easier to handle multiple files and future data additions.

- Initial Insights through EDA: The EDA process helped identify trends, patterns, and anomalies, providing a basis for more detailed future analysis.

Combined Correlation Heatmap

Summary and Challenges in Feature Engineering

The dataset consists of trip records from 2020 to 2023, capturing attributes such as pickup, dropoff times, trip distance, fare details, and other relevant features like payment type, passenger count, and location IDs. The primary goal of the data cleaning process was to standardize this information by handling missing values, dropping unnecessary columns, and ensuring consistent naming across all attributes.

Feature engineering for this dataset presents a few challenges. One potential issue is the handling of outliers, such as extremely high or low fare amounts, distances, or unusual pickup and dropoff times. Additionally, converting categorical data like payment types into meaningful numerical features which may require careful encoding, especially if some values appear sporadically throughout the dataset. Another challenge will be the temporal features, like rush hour vs. off-peak or weekdays vs. weekends which can require additional time-based transformations. We

will need to be cautious to avoid data leakage when creating features that might use information not available at prediction time.

During the feature engineering phase, I will have to carefully consider what will be needed to handle these complexities to ensure that the derived features are both informative and robust, leading to better model performance.

Project Milestone 4

Col name	Data type	Feature engineering treatment
trip_miles	Double	numeric feature
tolls	Double	numeric feature
sales_tax	Double	numeric feature
congestion_surcharge	Double	numeric feature
trip_duration	Double	Scaled using VectorAssembler
tips	Double	Target variable (label)

Code Description

1. Reading Cleaned Data and Processing
- The cleaned data is read from the /cleaned folder using the SparkSession.

- Feature engineering is performed to derive trip_duration as a critical input feature.

2. Feature Engineering
Feature engineering is performed using the feature_engineering function. The following transformations are applied:

Calculate Trip Duration:

- The duration of each trip is calculated by taking the difference between dropoff_datetime and pickup_datetime using unix_timestamp.

- The original timestamp columns are dropped to save memory and streamline processing.

- A VectorAssembler combines input columns into a single feature vector for the model.

- The chosen columns (trip_miles, tolls, sales_tax, congestion_surcharge, trip_duration) represent key aspects of a trip that may influence the tip amount.

3. Train/Test Split

- Data is split into training and testing subsets for model building and evaluation.

- Utilized randomSplit which divides the data into 80% for training and 20% for testing.

4. Modeling

Linear Regression from PySpark's MLlib library is used for its simplicity and interpretability.

The model predicts tips (the target variable) using the combined feature vector (features).

A regularization parameter (regParam=0.01) is set to prevent overfitting.

5. Validation and Evaluation

The trained model is validated using:

- **Mean Squared Error (MSE):** Measures the average squared difference between actual and predicted tips.
- **Mean Absolute Error (MAE):** Calculates the average absolute difference between actual and predicted tips.
- **R-squared (R²):** Indicates how well the model explains variance in the data (closer to 1 is better).

6. Saving Outputs

- Feature-Engineered Data: Processed data for each month is saved to the /trusted folder in Parquet format.
- Trained Models: The trained model is saved to the /models folder in the bucket.

Appendix B:

EDA Source Code:

```python
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns


# Set display options for better readability

pd.set_option('display.float_format', '{:.2f}'.format)

pd.set_option('display.width', 1000)

%matplotlib inline


# Function to load data from multiple files

def load_data(file_paths, file_type='csv'):

    df_list = []

    for file_path in file_paths:

        if file_type == 'csv':

            df = pd.read_csv(file_path)

        elif file_type == 'parquet':

            df = pd.read_parquet(file_path, engine='pyarrow', storage_options={'anon': True})

        elif file_type == 'json':

            df = pd.read_json(file_path, lines=True)

        else:

            raise ValueError("Unsupported file type")


        df_list.append(df)
```

```python
    # Concatenate all DataFrames into a single DataFrame
    return pd.concat(df_list, ignore_index=True)


# Function to perform EDA (without visualizations)
def perform_eda(df):
    # Basic Information
    print("Basic Data Info:")
    print(df.info())


    # Descriptive Statistics
    print("\nDescriptive Statistics:")
    print(df.describe())


    # Check for Missing Values
    print("\nMissing Values per Column:")
    print(df.isnull().sum())


    # Replace or Drop Missing Values Example
    df.fillna({'passenger_count': 0, 'RatecodeID': 1, 'congestion_surcharge': 0, 'airport_fee': 0},
inplace=True)
    df.dropna(subset=['fare_amount', 'trip_distance'], inplace=True)


    # Convert to DateTime
    if 'tpep_pickup_datetime' in df.columns and 'tpep_dropoff_datetime' in df.columns:
        df['tpep_pickup_datetime'] = pd.to_datetime(df['tpep_pickup_datetime'])
        df['tpep_dropoff_datetime'] = pd.to_datetime(df['tpep_dropoff_datetime'])
```

```python
        return df


# Function to process data in batches and combine afterward
def process_in_batches_and_combine(file_paths, batch_size=3, file_type='parquet'):
    num_files = len(file_paths)
    batch_results = []

    # Process files in batches
    for i in range(0, num_files, batch_size):
        batch_files = file_paths[i:i + batch_size]
        print(f"Processing batch: {batch_files}")

        # Load and process batch
        df_batch = load_data(batch_files, file_type)
        processed_df = perform_eda(df_batch)

        # Save batch result to a list
        batch_results.append(processed_df)

    # Combine all batches into one final DataFrame
    final_df = pd.concat(batch_results, ignore_index=True)
    return final_df


# Function to generate combined visualizations
def generate_combined_visualizations(df):
```

```python
sns.set_style("white")


# Histogram for Trip Distance
if 'trip_distance' in df.columns:

    plt.figure(figsize=(10, 5))

    sns.histplot(df['trip_distance'], bins=30)

    plt.xlim(0, 50)  # Limit to a reasonable range for better visibility

    plt.title('Combined Trip Distance Distribution')

    plt.xlabel('Trip Distance (miles)')

    plt.show()


# Scatter Plot
if 'fare_amount' in df.columns and 'trip_distance' in df.columns:

    plt.figure(figsize=(10, 5))

    sns.scatterplot(x='trip_distance', y='fare_amount', data=df)

    plt.xlim(0, 50)  # Limit the x-axis to trips less than 50 miles for better scaling

    plt.yscale('log')  # Apply a logarithmic scale to the y-axis

    plt.title('Combined Trip Distance vs Fare Amount')

    plt.xlabel('Trip Distance (miles)')

    plt.ylabel('Fare Amount (log scale)')

    plt.show()


# Correlation Heatmap
numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns

plt.figure(figsize=(8, 6))

sns.heatmap(df[numeric_cols].corr(), annot=True, cmap="YlGnBu")
```

```python
    plt.title('Combined Correlation Heatmap')

    plt.show()


# Generate file paths for the years 2020 to 2023 in Parquet format

file_paths = [

    f"https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_{year}-{str(month).zfill(2)}.parquet"

    for year in range(2020, 2024) for month in range(1, 13)

]

file_type = 'parquet'


# Process data in batches and combine into one final DataFrame

final_df = process_in_batches_and_combine(file_paths, batch_size=3, file_type=file_type)


# Generate combined visualizations after processing all data

generate_combined_visualizations(final_df)
```

Appendix C:

Data Cleaning Code:

```python
# Import necessary libraries
import pandas as pd
from google.cloud import storage
import os

# Set up Google Cloud Storage client
client = storage.Client()
bucket_name = "my-bigdata-project-aa"
landing_folder = "landing"
cleaned_folder = "cleaned"

# Function to load data from GCS
def load_data_from_gcs(file_path):
    bucket = client.get_bucket(bucket_name)
    blob = bucket.blob(file_path)
    with blob.open("rb") as file:
        return pd.read_parquet(file)

# Function to clean data
def clean_data(df):
    # Remove spaces from column names and standardize to lowercase
    df.columns = [col.strip().replace(" ", "_").lower() for col in df.columns]

    # Drop unneeded columns
    unneeded_columns = ['store_and_fwd_flag', 'shared_request_flag', 'shared_match_flag',
'access_a_ride_flag']
    df.drop(columns=[col for col in unneeded_columns if col in df.columns], inplace=True, errors='ignore')

    # Fill or drop missing values
    df.fillna({
        'passenger_count': 0,
        'ratecodeid': 1,
        'congestion_surcharge': 0,
        'airport_fee': 0
    }, inplace=True)
    df.dropna(subset=['base_passenger_fare', 'trip_miles'], inplace=True)  # Replace with appropriate
columns

    # Convert data types for efficiency and consistency
    df = df.astype({
```

```python
        'hvfhs_license_num': 'category',
        'dispatching_base_num': 'category',
        'pulocationid': 'Int64',
        'dolocationid': 'Int64',
        'trip_miles': 'float',
        'base_passenger_fare': 'float'
    })

    # Remove records with invalid data
    df = df[(df['trip_miles'] > 0) & (df['base_passenger_fare'] > 0)]

    # Handle outliers by capping at the 99th percentile
    if 'base_passenger_fare' in df.columns:
        fare_cap = df['base_passenger_fare'].quantile(0.99)
        df = df[df['base_passenger_fare'] <= fare_cap]
    if 'trip_miles' in df.columns:
        miles_cap = df['trip_miles'].quantile(0.99)
        df = df[df['trip_miles'] <= miles_cap]

    return df

# Function to save cleaned data back to GCS
def save_cleaned_data_to_gcs(df, output_file_path):
    bucket = client.get_bucket(bucket_name)
    output_blob = bucket.blob(output_file_path)
    output_blob.upload_from_string(df.to_parquet(index=False), content_type="application/octet-stream")
    print(f"Cleaned data saved to {output_file_path}")

# Main function to process all files
def process_files():
    for year in range(2020, 2024):
        for month in range(1, 13):
            file_path = f"{landing_folder}/fhvhv_tripdata_{year}-{month:02}.parquet"
            output_path = f"{cleaned_folder}/fhvhv_tripdata_{year}-{month:02}.parquet"
            try:
                # Load data
                print(f"Processing {file_path}")
                df = load_data_from_gcs(file_path)

                # Clean data
                df_cleaned = clean_data(df)

                # Save cleaned data
                save_cleaned_data_to_gcs(df_cleaned, output_path)
```

```python
        except Exception as e:
            print(f"Failed to process {file_path}: {e}")

# Run the processing function
process_files()
```

Appendix D:

```python
#importing necessary libraries

from pyspark.sql import SparkSession

from pyspark.sql.functions import unix_timestamp, col

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.regression import LinearRegression

from pyspark.ml.evaluation import RegressionEvaluator


# Initialize Spark session with memory optimization

spark = SparkSession.builder \

    .appName("Taxi Data Batch Processing and Model Training") \

    .config("spark.executor.memory", "8g") \

    .config("spark.executor.cores", "2") \

    .config("spark.executor.instances", "4") \

    .config("spark.speculation", "true") \

    .getOrCreate()


# Input and output paths

input_folder_path = "gs://my-bigdata-project-aa/cleaned/"

trusted_folder_path = "gs://my-bigdata-project-aa/trusted/"

model_folder_path = "gs://my-bigdata-project-aa/models/"


# List of files to process (2020-2023 all months)

file_list = [f"fhvhv_tripdata_{year}-{month:02d}.parquet"
```

```python
            for year in range(2020, 2024) for month in range(1, 13)]


# Process files individually

for file_name in file_list:

    print(f"Processing file: {file_name}")

    try:

        # Read file from `cleaned` folder

        file_path = f"{input_folder_path}{file_name}"

        df = spark.read.parquet(file_path)


        # Select relevant columns

        required_columns = [

            "pickup_datetime", "dropoff_datetime", "trip_miles", "tolls",

            "sales_tax", "congestion_surcharge", "tips"

        ]

        df = df.select(*required_columns)


        # Feature engineering: Calculate trip duration

        df = df.withColumn(

            "trip_duration",

            unix_timestamp(col("dropoff_datetime")) - unix_timestamp(col("pickup_datetime"))

        ).drop("pickup_datetime", "dropoff_datetime")


        # Save processed file directly into the `trusted` folder
```

```python
        trusted_file_path = f"{trusted_folder_path}{file_name}"

        df.coalesce(1).write.mode("overwrite").parquet(trusted_file_path)

        print(f"Saved processed file to: {trusted_file_path}")


    except Exception as e:

        print(f"Error processing file {file_name}: {e}")

        continue  # Skip to the next file if there's an issue


# Combine all processed files

print("Combining all processed files from the trusted folder...")

try:

    combined_df = spark.read.parquet(f"{trusted_folder_path}*.parquet")

    print("All files combined successfully.")

except Exception as e:

    print(f"Error combining files: {e}")

    exit(1)


# Prepare data for model training

assembler = VectorAssembler(

    inputCols=["trip_miles", "tolls", "sales_tax", "congestion_surcharge", "trip_duration"],

    outputCol="features"

)

combined_df = assembler.transform(combined_df).select("features", "tips")
```

```python
# Train-test split

train_data, test_data = combined_df.randomSplit([0.8, 0.2], seed=42)


# Train Linear Regression model

print("Training Linear Regression model...")

lr = LinearRegression(featuresCol="features", labelCol="tips", regParam=0.01)


# Fit the model on training data

model = lr.fit(train_data)


# Save the model

model_path = f"{model_folder_path}taxi_tip_prediction_model"

try:

    model.write().overwrite().save(model_path)

    print(f"Model saved to: {model_path}")

except Exception as e:

    print(f"Error saving the model: {e}")


# Evaluate the model

print("Evaluating the model...")

predictions = model.transform(test_data)


evaluator = RegressionEvaluator(labelCol="tips", predictionCol="prediction")
```

```python
mse = evaluator.evaluate(predictions, {evaluator.metricName: "mse"})

mae = evaluator.evaluate(predictions, {evaluator.metricName: "mae"})

r2 = evaluator.evaluate(predictions, {evaluator.metricName: "r2"})


print("Model Evaluation Metrics:")

print(f"Mean Squared Error (MSE): {mse}")

print(f"Mean Absolute Error (MAE): {mae}")

print(f"R-squared (R2): {r2}")
```